SUBMITTED TO: SPECIAL ISSUE ON FAULT-TOLERANT COMPUTING OF THE IEEE TRANSACTIONS ON COMPUTERS

(2)

PROGRESSIVE RETRY FOR SOFTWARE FAILURE RECOVERY IN MESSAGE-PASSING APPLICATIONS

Yi-Min Wang* Yennun Huang* W. Kent Fuchs† Chandra Kintala*

*AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974

†Coordinated Science Laboratory
University of Illinois
Urbana, IL 61801

Abstract

In this paper, we describe a method of execution retry for bypassing software faults in message-passing applications. Based on the techniques of checkpointing and message logging, we demonstrate the use of message replaying and message reordering as two mechanisms for achieving localized and fast recovery. Our approach gradually increases the rollback distance and the number of affected processes when a previous retry fails, and is therefore named *progressive retry*. An implementation as reusable modules to provide low-cost application-level software fault tolerance is described. Examples from our experience with telecommunications software systems are given to illustrate the benefits of the scheme.

Key words: software fault tolerance, failure recovery, checkpointing and rollback, message logging, message replaying, message reordering

The research of the third author was supported in part by the Department of the Navy and managed by the Office of the Chief of Naval Research under Contract N00014-91-J-1283, and in part by the National Aeronautics and Space Administration (NASA) under Grant NASA NAG 1-613, in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS).

[1]A brief earlier version of this paper was presented at the *IEEE Fault-Tolerant Computing Symposium*, June 1993.

1 Introduction

For computer systems designed to provide continuous services to customers, availability is an important performance measure. In such systems, software failures have been observed to be the current major cause of service unavailability [1,2]. Residual software faults due to untested boundary conditions, unanticipated exceptions and unexpected execution environments have been observed to escape the testing and debugging process and, when triggered during program execution, cause service interruption [3]. It is therefore desirable to have effective on-line retry mechanisms for automatically bypassing software faults and recovering from software failures in order to achieve high availability [4–7].

A software fault in a program is triggered when the program is executed in a certain environment and supplied with a certain set of input data. Therefore, the resulting error can potentially be recovered by one of the following three approaches: (1) executing a different program which implements the same function but does not contain the same fault, called the *design diversity* approach; (2) executing the same program on a different set of data obtained through acceptable transformation, called the *data diversity* approach [8,9]; (3) executing the same program in a different environment, called the *environment diversity* approach.

N-version programming [10] and *recovery blocks* [11] are two well-known techniques employing design diversity. While the former executes N different implementations at the same time and votes on the results, the latter invokes an alternate only when the previous one fails the acceptance test. Design diversity is effective in masking software faults if the assumption that different implementations of the same function will not contain the same fault is valid.

N-copy programming and *retry blocks* [9] are the counterparts of the above two techniques, which adopt the data diversity approach. Instead of executing different programs on the same data set, such techniques execute the same program on different sets of data obtained through acceptable data reexpression algorithms. Data diversity is effective for those cases in which reexpression algorithms can be found to express the data points from the failure region [9] in terms of those outside that region.

Several studies [2, 12, 13] have shown that many software failures in production systems behave in a transient fashion, and so the simplest way to recover from such failures is to restart the system, an approach that we call environment diversity. The term *Heisenbug* [1] has been used to refer to the software faults causing transient failures, while the term *Bohrbug* refers to those software faults with deterministic behaviors. *Process pairs* - primary and backup running the same program [5, 14, 15] - provide a way of bypassing Heisenbugs. The periodic checkpoint messages sent by the primary enable the backup to maintain approximately the same state as the primary so that it can quickly take over when the primary fails.

In this paper, we describe a *progressive retry* technique for software failure recovery in message-passing applications[2]. The target applications are continuously-running software systems for which fast recovery is essential and a reasonable amount of run-time overhead may not result in noticeable service quality degradation. Many telecommunications systems fall into this category. There are several reasons that fast recovery is desirable in applications requiring high availability. In the cases where service quality is judged at the user interface level, small "computer down time" involving only a small number of processes may be translated into zero "service down time." Most importantly, when the prolonged unavailability of one part of the system may trigger the boundary conditions in other parts of the system, localized and fast recovery can reduce the possibility of cascading failures which may lead to a catastrophe.

Our progressive retry technique is based on checkpointing, rollback, message replaying and message reordering. The goal is to limit the *scope of rollback*: the number of involved processes as well as total rollback distance. The approach consists of several retry steps and gradually increases the scope of rollback when a previous retry fails. Progressive retry has the following three main contributions. First, although Heisenbugs can be bypassed by simply restarting the system, faster recovery can often be achieved by locally "exploiting" new environments through

[2]We will focus on error recovery in this paper; the issue of error detection is considered elsewhere [16].

message replaying or "simulating" possible environments through message reordering. Second, although Heisenbugs may constitute the majority of software faults in some environments[12], our experience indicates that many network failures resulting in extensive system outages have been caused by rarely triggered Bohrbugs. As an extreme example, 99% Heisenbugs may contribute only 1% of total down time because they can be easily bypassed, while the remaining 1% Bohrbugs may cause 99% of total down time. Non-transient failures may occur when some input messages from outside the system cannot be rolled back and the processing of such messages always leads to the same failure. Message reordering provides an opportunity of introducing data diversity through changing the sequence of input messages to bypass such Bohrbugs. Finally, our progressive retry technique can be implemented in reusable software modules to be incorporated into existing systems and shared by many different applications [16].

This paper is organized as follows. Section 2 gives the logical checkpoint model for rollback recovery based on checkpointing as well as message logging; Section 3 describes the progressive retry technique; Section 4 gives several examples to demonstrate the capability of progressive retry for bypassing software faults; our current implementation is described in Section 5, and possible extensions and limitations are discussed in Section 6.

2 Checkpointing and Rollback Recovery

Numerous checkpointing and rollback recovery techniques have been proposed in the literature to recover from transient *hardware* failures. *Uncoordinated checkpointing* schemes [17, 18] allow maximum process autonomy and general nondeterministic execution, but suffer from potential domino effects [11]. *Coordinated checkpointing* schemes [19–21] eliminate the domino effect by sacrificing a certain degree of process autonomy and by paying the cost of extra coordination messages. A *lazy checkpoint coordination* technique [22] has been proposed as a mechanism for bounding rollback propagation and providing a flexible trade-off between run-time coordination overhead and recovery efficiency. *Log-based recovery* [23–29] assumes the *piecewise deterministic* *model* [30] and employs message logging and replaying to avoid domino effects.

In this paper, we apply the above techniques to *software* failure recovery. More specifically, we use log-based recovery to localize the retry and use message comparison (as described later) to verify the assumption of piecewise deterministic execution. When all localized retries fail to bypass the software faults, a globally consistent set of checkpoints obtained through lazy checkpoint coordination is used for large-scope rollback. In the remainder of this section, we use the example in Fig. 1 to introduce the notion of *logical checkpoints* for reasoning about the two primary mechanisms for bypassing software faults, namely, message replaying and message reordering.

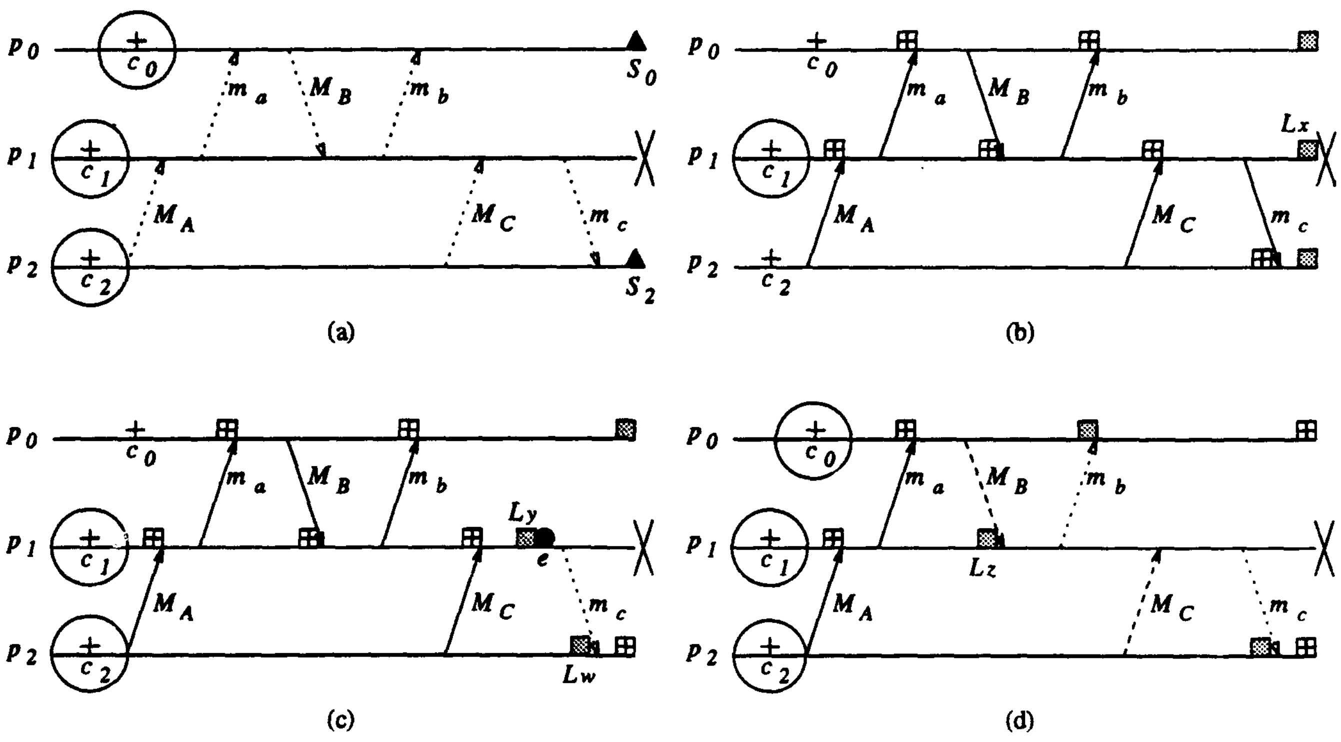

Figure 1: (a) Rollback recovery without the piecewise deterministic model; (b) message replaying, (c) message replaying with an interrupting nondeterministic event, and (d) message reordering under the piecewise deterministic model. (Circled checkpoints are used for the recovery.)

Without the assumption of piecewise determinism (i.e., process execution between any two consecutive message receipts is deterministic), when process p_1 in Fig. 1(a) initiates a rollback to checkpoint c_1 from the point marked "X", it *unsends* message m_c and should request p_2 to roll back as well in order to *unreceive* m_c, i.e., to nullify the effect of m_c which has been revoked. Otherwise, the system would have been left in an *inconsistent* state because c_1 and S_2 together show that m_c is "received but not yet sent." Similarly, p_0 also needs to roll back from S_0 in order to *unreceive* m_a and m_b. As a result, the latest consistent set of states for the system to roll back, called the *recovery line*, consists of c_0, c_1 and c_2. In this case, all six messages are *unsent* and *unreceived* and should be discarded.

Now suppose the assumption of a piecewise deterministic model is valid, and the message logs (*message contents* and *processing order*) for all six messages are available. Then p_1 can restart from the *physical checkpoint* c_1 and replay messages M_A, M_B and M_C in exactly the original order to deterministically reconstruct the state immediately before the rollback. This can be modeled as having an additional *logical checkpoint* L_x immediately before the "X" mark, as shown in Fig. 1(b). Therefore, although p_1 physically rolls back to c_1, it logically rolls back to L_x and so does not *unsend* any message. The recovery line in this case consists of the three logical checkpoints shown as shaded squares in (b). No rollback of p_0 or p_2 is necessary.

Although piecewise determinism can be exploited to limit the scope of rollback, it may not be valid throughout the entire execution. For example, some nondeterministic events may depend on functions of real time and simply cannot be recorded and replayed. Suppose such an event e exists in p_1's execution before message m_c was sent out, as shown in Fig. 1(c). Process p_1 can only reconstruct its state up to L_y which is immediately before event e, and all the logical checkpoints beyond that (in this case, L_x) become unavailable. The rollback of p_1 to L_y then *unsends* m_c and results in the rollback of p_2 to L_w to *unreceive* m_c.

Rollback propagation can also result from the processing order of M_B and M_C being lost upon a failure or being intentionally discarded to allow reordering. The latest available logical checkpoint of p_1 then becomes L_z (Fig. 1(d)) because any state beyond that cannot be deterministically reconstructed. As a result, processes p_0 and p_2 are required to roll back as well to *unreceive* m_b

and m_c, respectively.

It is important to recognize which messages can be reordered and which messages must be replayed in their original order to ensure correctness. For the example in Fig. 1(d), p_1 must replay message M_A in the original order after restarting from c_1 to reach the state L_z which is part of the recovery line. Similarly, message m_a must be deterministically replayed[3] by p_0. Such messages are drawn in solid lines. In contrast, messages M_B and M_C are recorded as "sent but not yet received" according to the recovery line. It is as though they are messages that are still traveling in the communication channels and therefore can arrive in an arbitrary order due to unknown transmission delay. Such messages, drawn in dashed lines, are called *in-transit messages*, and they are the messages that can be reordered. Finally, messages like m_b and m_c, which are *unsent* and *unreceived* with respect to the recovery line, are no longer valid. They become *orphan messages*, drawn in dotted lines, and should be discarded.

3 Progressive Retry

For the purpose of presentation, we assume that (1) every message is logged before delivery to the application process[4]; (2) only direct dependency [26,31] resulting from processing each individual message is recorded, no transitive information is propagated; (3) centralized recovery line computation[5] is performed based on the global dependency information collected by the process which initiates the garbage collection or recovery procedure; (4) both sender logging (of the messages sent) and receiver logging (of the messages received) are employed.

We will use the example checkpoint and communication pattern shown in Fig. 2 to illustrate progressive retry. Each step may involve multiple retries, depending on the application.

[3]For simplicity, we will use "replay" to mean "deterministically replay" throughout the paper.

[4]Volatile message logs lost upon a failure in asynchronous (optimistic) logging protocols can be modeled as unavailable logical checkpoints

[5]A synchronized, distributed algorithm has been proposed by Sistla and Welch [27]; an asynchronous, distributed algorithm can be found in Strom and Yemini's paper [25].

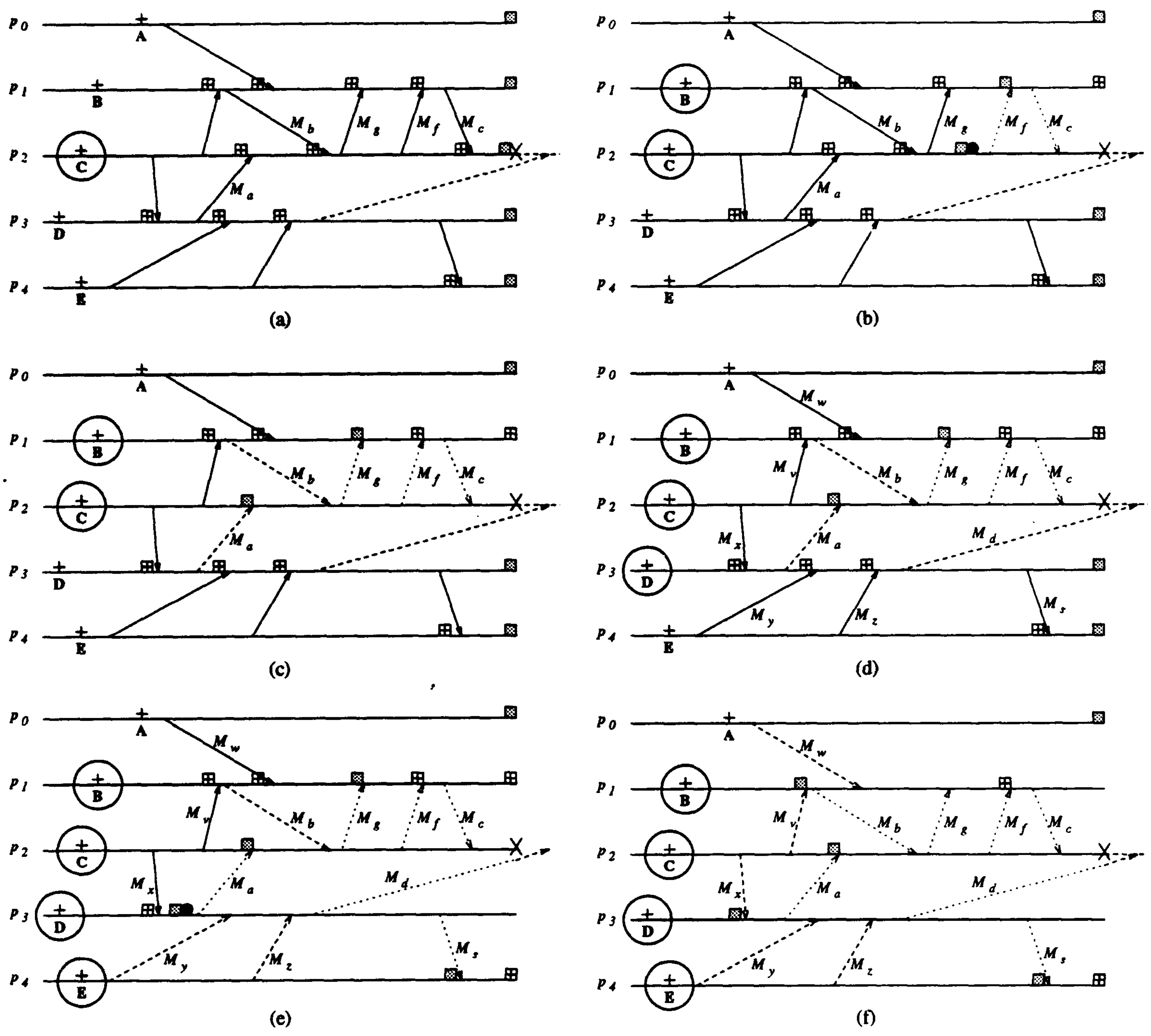

Figure 2: Progressive retry. (a) and (b) Step 1: receiver replaying; (c) Step 2: receiver reordering; (d) and (e) Step 3: sender replaying; (f) Step 4: sender reordering.

Step 1 - Receiver replaying

In Fig. 2(a), when p_2 detects an error (marked "X"), it first initiates a local recovery by rolling back to checkpoint C and deterministically replaying the message logs. Because every message is logged before processing, message logs for M_a, M_b and M_c must be available and allow p_2 to reconstruct the state up to the point it detected the error. In some cases, transient failures may be caused by some environmental factors (such as mutual exclusive conflicts, resource unavailability, unexpected signals, etc.) which will simply disappear after the recovery, and Step-1 retry may succeed. If the reexecution still leads to the same error, the checkpoint and message log information is copied to a trace file for off-line debugging.

Fig. 2(b) illustrates another possible scenario. After the rollback, when p_2 is about to reexecute the sending of message M_f, it compares the newly generated message to the corresponding one in the sender log from previous execution and detects a discrepancy. Process p_2 then realizes that some nondeterministic event e has interrupted the state reconstruction, and therefore all the messages sent beyond that event need to be *unreceived*. The new recovery line (shaded squares) shown in Fig. 2(b) reflects the resulting rollback propagation. Such a departure from previous execution offers an additional opportunity for bypassing a software fault causing a transient failure..

Step 2 - Receiver reordering

When Step-1 retry fails, it is likely that the failure is caused by the messages from other processes. Since the goal of progressive retry is to limit the scope of rollback, p_2 first tries to locally simulate other possible scenarios regarding the messages without actually involving the senders. Because one in general cannot predict the transmission delays of the incoming communication channels, the original software could not have been written assuming a particular ordering. Therefore, message reordering can be used to simulate different message arrival orders.

For example, in Fig. 2(c), p_2 decides to reorder the messages M_a, M_b and M_c in its receiver log. But once the processing order of these three messages is changed, the last three logical checkpoints of p_2 (shown in Fig. 2(a)) become unavailable and the new recovery line needs to be computed to determine which messages can be reordered. With respect to the new recovery line, only M_a and M_b are in-transit messages available for reordering; message M_c, as well as M_g and M_f, now become orphan messages and should be discarded.

There are several potentially useful algorithms for reordering the message logs. Random reordering can be used when no knowledge about the possible cause of the software failure is available. If the failure is possibly due to the interleaving of messages from different processes, reordering by grouping the messages from the same process together may be useful. If the software fault might have been triggered by exhausting all available resources, reordering the messages so that every resource is freed at the earliest possible moment can often bypass the boundary condition.

Step 3 - Sender replaying

When p_2 fails to bypass the fault through locally replaying and reordering its receiver log, it discards all the received messages after checkpoint C and requests the senders of those messages to participate in the Step-3 retry. The senders roll back and first try to replay their receiver logs up to the recovery line. Even though exactly the same messages as those sent before the failure may be generated, there are still two possibilities that are useful in bypassing the fault. First, when the number of messages is large and the cause of the failure is totally unknown, the correct message sequences may not be covered in the limited number of "forced" reordering in Step 2. The senders' resending these messages provides another opportunity for obtaining a correct sequence through "natural" reordering. Second, when a software fault is triggered by some unexpected delay in the delivery of certain messages (such as M_d in Fig. 2(d)), Step-2 retry may be insufficient because such messages are not yet in the receiver log. If the environmental factors that caused the unexpected delay have disappeared during reexecution, resending such messages allows them to be included in the reordering (either "natural" or "forced") to bypass the fault.

As in Step 1, some nondeterministic events may interrupt the state reconstruction through message replaying. Fig. 2(e) illustrates a possible scenario where p_3, during its reexecution, detects a discrepancy between the message M_a in its previous sender log and the new message it has generated. The different execution path beyond the nondeterministic event e then allows a potentially more effective retry (because different messages are now sent to p_2) at the expense of involving one additional process p_4 in the retry.

Step 4 - Sender reordering

When sender replaying still cannot bypass the fault, it is suspected that the error might have originated at one of the senders, which consequently sent out erroneous messages and caused the receiver to fail. Therefore, the sender should take a different execution path in order to revoke the erroneous messages.

In the example shown in Fig. 2(f), p_1 and p_3 are requested to roll back to the logical checkpoints before any message in p_2's receiver log was sent[6]. Process p_1 then reorders M_v and M_w, and p_3 reorders M_x, M_y and M_z so that the possibly erroneous message (M_a or M_b), which resulted in p_2's failure, can be corrected.

Step 5 - Large-scope rollback retry

When all previous small-scope retries fail, a large-scope rollback is initiated. The simplest way is to restart the application. However, the potentially costly state reinitialization process may result in extensive service outage which cannot be tolerated for high-availability applications. A globally consistent set of checkpoints can be used in such cases to achieve faster rollback retry. Because of

[6]Process p_2 in this case is similar to the *exhausted importer* in the Programmer-Transparent Coordination scheme developed by Kim *et al.* [32]. It accuses p_1 and p_3 of exporting erroneous information which is responsible for its own failure, after unsuccessful local retries.

the issue of checkpoint consistency as described in Section 2, checkpoint coordination is necessary for obtaining a globally consistent set of checkpoints. Chandy and Lamport's distributed snapshot algorithm [19] can be used to achieve efficient checkpoint coordination [33]. The coordinator broadcasts a `marker` message to initiate the coordination. Each process takes a checkpoint upon receiving a `marker` message and increments its current *checkpoint interval number*. The above number is piggybacked on every message M sent so that the receiver p_r can decide to take a checkpoint before processing M for the purpose of coordination if the piggybacked number is greater than p_r's current checkpoint interval number.

An alternative to achieving a consistent set of checkpoints is to use lazy checkpoint coordination [22, 34]. For many applications requiring high availability, it is often desirable to let each process decide where to take checkpoints so that only *critical data* [16, 35], instead of the entire process state, needs to be saved and restored. If checkpoint coordination is initiated for every checkpoint taken, there will be an unnecessarily large number of checkpoints. Another observation is that the communication patterns of many practical applications are not very complicated. Therefore, it is quite possible that two checkpoints taken independently by two processes are "naturally" consistent without the need of any coordination. This leads to the concept of lazy coordination: optimistically assuming that checkpoints which need to be consistent will be naturally consistent, and paying the overhead of coordination only when the assumption is about to fail. An integer parameter Z called *laziness* which controls the coordination frequency is specified and is known to all processes in the system. The consistency criterion is that, for any integer n, checkpoints $\#nZ$ of all processes must form a consistent set of checkpoints. Such a criterion can be enforced by the receivers of certain messages taking additional checkpoints before processing the messages, based on the piggybacked checkpoint interval numbers. Figure 3 gives an example where a "+" sign represents a *basic checkpoint* initiated independently by each individual process, and a circled "+" sign represents an *induced checkpoint* taken for the purpose of coordination. Figure. 3(b) illustrates the case of $Z = 1$: a checkpoint is induced whenever a receiver detects that a sender's checkpoint interval number is greater than its own. The case of $Z = 2$ is shown in Fig. 3(c); the only induced checkpoint is taken when p_1 at its checkpoint interval #1 is about to process a message sent from p_0's checkpoint interval #2, which would have violated the consistency criterion.

The parameter Z provides a trade-off among several things. The advantages of choosing a larger Z are (1) lower coordination overhead, i.e., less induced checkpoints (as demonstrated in Fig. 3(c)) and (2) potentially more effective retry for recovering from errors with large latencies. Potential disadvantages are (1) larger rollback distance which implies longer service unavailability; (2) outputs directed to outside the system either need to wait longer before they can be released, or they can be released immediately but more of them might be revoked as a result of rollback; (3) larger space overhead for storing more checkpoints and message logs. Therefore, in practice, the choice of laziness will be constrained by run-time overhead, maximum error latencies, recovery deadlines, output delays (or penalties for revoking outputs) and stable storage space overhead, depending on the requirements of the application.

4 Experience and Examples

In this section, we describe four examples with software faults that can be bypassed by Steps 1-4 to demonstrate the capability of progressive retry to achieve localized and fast recovery. To simplify the description, we have abstracted only the software components which contribute to the failures.

Step-1 retry

A telecommunication billing system consists of several processes accessing large shared data structures in response to the incoming request messages for data access. There is only one writer process and the others are all reader processes. Because no locking mechanism is provided to guarantee mutual-exclusive accesses, there is a small probability that a reader may be accessing a data structure while the writer is updating it (e.g., manipulating the pointers for inserting a new data

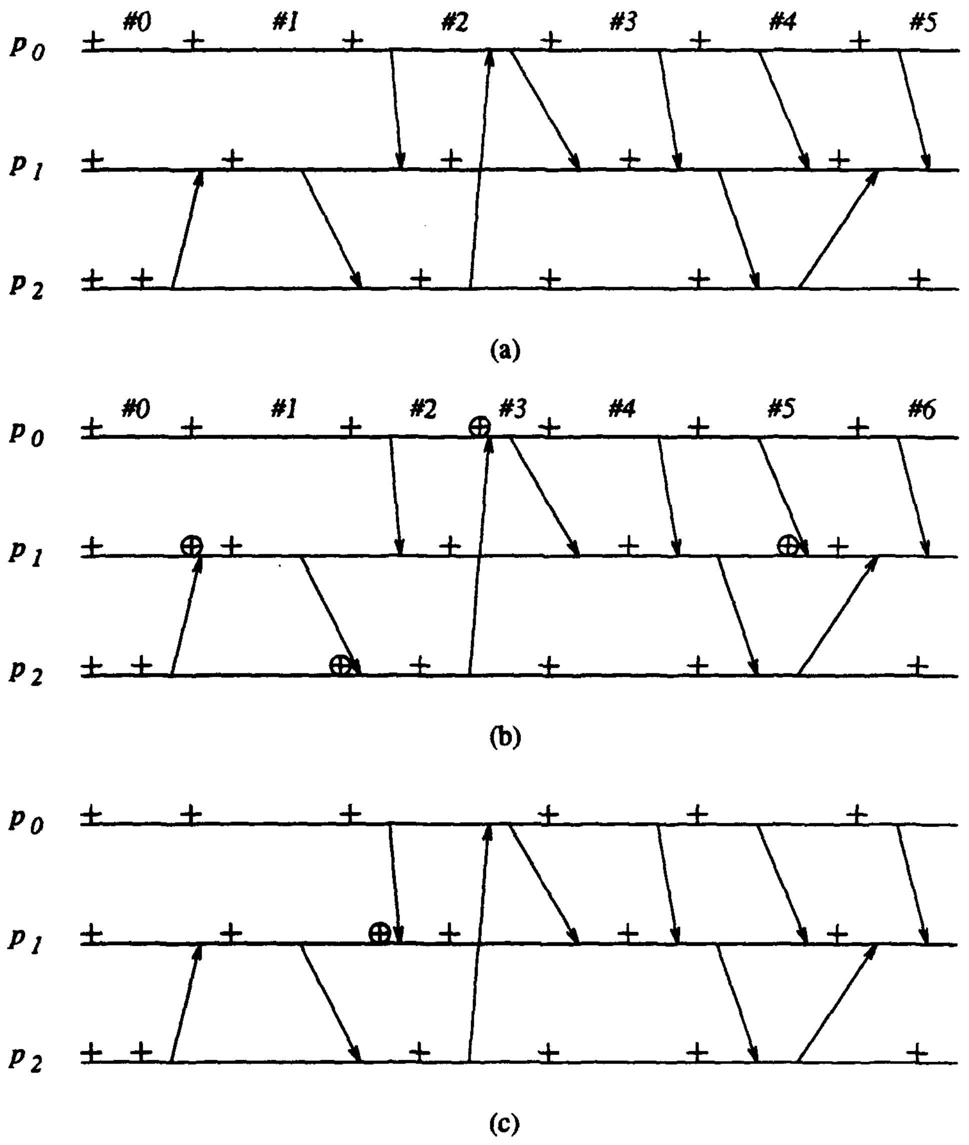

Figure 3: (a) Checkpoint and communication pattern without any coordination; lazy checkpoint coordination with (b) $Z = 1$ and (c) $Z = 2$.

node). In such a case, the reader receives a segmentation violation fault and is then recovered by a watchdog process. Once the reader is restarted and replays the request message for accessing the same data structure again, the read operation succeeds because the writer has finished the update.

This kind of error occurs once every few days. Whenever it happens, Step-1 retry can quickly correct the error. An alternative (standard) way of dealing with this problem would be to use a locking mechanism. However, using coarse-grain locks can result in unnecessary blocking of the reader processes, and using fine-grain locks can incur large performance degradation and introduce additional software complexity. Therefore, the billing system has relied on message replaying as an alternative to locking for dealing with the concurrency problem. The approach is feasible in this case because mutual-exclusive conflicts only occur rarely and they can be detected when they occur.

Step-2 retry

The nDFS [36] file system provides on-line replication of critical files. It contains three main processes: Local Replication (LR) process, Backup Read (BR) process and Backup Processing (BP) process. All `open`, `write` and `close` system calls on the primary node are intercepted by a shared library and passed to process LR, which then multiplexes system calls from different applications and sends update messages to process BR on the backup node. Process BR is responsible for logging the update messages and placing them into a buffer where process BP can retrieve the messages and perform the updates. Since the number of available file descriptors for process BP is limited and each application process could open many files at the same time, process BP may run out of file descriptors. Therefore, it has to keep track of how many files are currently open. A boundary condition occurs when all file descriptors are used. Process BP then searches for an open file descriptor with the earliest access time and closes that file.

A software bug exists in the search procedure so that once process BP enters the boundary condition, the search never finishes and the process BP hangs up. Process BP implements checkpointing and message logging and has an external hang-up detection mechanism. When it enters the boundary condition, the failure is detected and the process is recovered by restoring the checkpointed state and reexecuting the logged messages.

The following example illustrates how Step-2 message reordering can bypass the boundary condition. Let `o1` command stand for opening file 1, `w1` command stand for writing data to file 1 and `c1` command stand for closing file 1. Suppose process BP can open at most 2 files at the same time. Then the following command sequence will cause process BP to enter the boundary condition when processing `o3` and hang up.

```
o1 o2 w1 w2 o3 w3 c1 c2
```

Suppose all the above commands are logged before the failure. When process BP is restarted, the command log may be reordered as follows:

```
o1 w1 c1 o2 w2 c2 o3 w3.
```

In this sequence, the boundary condition never occurs and therefore the reexecution of the command log succeeds.

Step-3 retry

In a cross-connection system, a *Channel Control Monitor* (CCM) process is used to keep track of the available channels in a switch. The CCM process receives messages from two other processes: a *Channel Allocation* (CA) process which sends the channel allocation requests and a *Channel Deallocation* (CD) process which sends the channel deallocation requests. A boundary condition for CCM occurs when all channels are used and the process receives additional allocation requests. In that case, a clean-up procedure is called to free up some channels or to block further requests. However, the clean-up procedure contains a software fault which could cause the process to crash.

The cross-connection system uses a daemon watchdog to detect process failures and employs checkpointing and message logging for recovery. The following example illustrates how progressive retry works in this system. Suppose the number of available channels is 5. The command `r2` stands for requesting two channels, and the command `f2` stands for freeing two channels. The following command sequence could cause the CCM process to crash because of the boundary error.

CA sends `r2  r3  r1`

CD sends `f2  f3  f1`

CCM receives `r2  r3  r1` and crashes

If the message `f2` is received and logged before CCM crashes, CCM will be able to recover by reordering the message logs. However, if CCM crashes before `f2` is logged, reordering messages `r2`, `r3` and `r1` (Step 2) will not help. In this case, the local recovery of CCM fails and CA and CD will be requested to resend their messages (Step 3). Because of the nondeterminism in operating system scheduling and transmission delay, the messages may arrive at CCM in a different order. For example, the message order can be

```
r2  f2  r3  f3  r1  f1
```

which does not lead to the boundary condition, and hence the retry succeeds.

Step-4 retry

We use the Signal Routing Points (SRPs) in a switching system as our fourth example. The responsibility of SRPs is to route data packets from the originating switch to the destination switch. Each SRP has a built-in overload control (OC) mechanism which, when the number of packets in its buffer exceeds a threshold, sends out OC messages to other SRPs to reduce incoming traffic as a precaution. Suppose in normal operation SRP-X can route data packets destined for a certain switch either through SRP-A or SRP-B. When SRP-X receives an OC message from SRP-A, it starts routing all such packets through SRP-B to avoid potential overload of SRP-A. But at the same time, the switch directly connected to SRP-B receives a sudden burst of service requests. These two kinds of traffic quickly fill the buffer of SRP-B and the process crashes before any overload control mechanism can be invoked, due to a software fault.

Local retries of SRP-B through message replaying and reordering cannot recover from the failure because they cannot bypass the overload condition. Step-3 retry involving SRP-X message replaying still leads to the same failure. SRP-X then initiates Step-4 retry by reordering the packets to be routed and the OC message from SRP-A. This effectively delays the processing of the OC message, and reduces the traffic through SRP-B to give SRP-B a chance to recover. The potential overload for SRP-A either never happens or it can be handled gracefully without causing a software failure.

5 Implementation

The progressive retry technique has been implemented in the `libft` library and the `watchd` daemon [16]. `Libft` library is a C/C++ library which provides functions to checkpoint process states and log messages; `watchd` daemon is a generic service process for detecting process failures and restarting failed processes.

`Libft` functions which are related to the progressive retry technique are `ftread()`, `ftwrite()`, `checkpoint()`, `recover()`, `ftreorder()`, `setlogfile()` and `in_recovery()`. In a normal operation, `ftwrite()` is used to send messages and create sender log files; `ftread()` is used to receive messages and create receiver log files. In a recovery state, `ftwrite()` does not send those messages which are generated by message replaying and have exact duplicates in the sender log file already; `ftread()` reads messages from the receiver log file, instead of a regular I/O channel. Function `in_recovery()` returns 0 if the calling process is in a normal state; otherwise, it returns 1 if the calling process is in Step-1, Step-3 or Step-5 retry, and it returns 2 if the calling process is in Step-2 or Step-4 retry which requires message reordering.

Functions `checkpoint()` and `recover()` are used to save and restore data from a stable storage. When an application process calls `checkpoint()`, it increments its checkpoint interval number and saves critical data onto a stable storage. Then, a message containing the checkpoint file name, the process id and the checkpoint interval number is sent to the `watchd` daemon. This information is then used by the `watchd` to replicate the checkpoint file onto one or more backup machines. When an uncoordinated checkpointing scheme with lazy checkpoint coordination is used, `watchd` invokes the garbage collection procedure when it detects that the checkpoint interval numbers of all processes have reached nZ and hence a new lazy-coordinated recovery line has been formed.

A message in a receiver log file contains six fields: message size, message content, sender id, sender's checkpoint interval number, sender's logical checkpoint number and reference id. Sender id is the number assigned by `watchd` at the start-up time of each application process that `watchd` monitors. Checkpoint interval number is incremented each time when a new checkpoint is taken; logical checkpoint number is incremented upon sending or receiving a new message. Reference id is given by function `ftwrite()` and is used for message reordering in recovery. The `ftreorder()` function provides a default message reordering routine which randomly reorders messages with different reference ids but maintains the partial order for messages with the same reference id. By specifying a reference id when `ftwrite()` is called, message dependency can be enforced if necessary. A message in a sender log file contains only four fields: message size, message content, receiver id and logical checkpoint number. The first two fields are used for message comparison; the last two fields are used to perform rollback propagation for determining the recovery line in a retry.

`Watchd` can also be configured to use a simplified implementation of progressive retry based a coordinated checkpointing scheme. Upon receiving a new checkpoint interval number, `watchd` sends a signal to all the other processes in the same application domain. Once the signal is received, each process turns on a flag and takes a checkpoint when the next `ftread()` is called. To decide which processes should be rolled back in a retry, `watchd` keeps track of the interprocess communication patterns. When process A receives a message (via `ftread()`) from process B for the first time in its current checkpoint interval, a message including the sender id and the receiver id is sent to the `watchd`. `Watchd` then uses this information to construct a communication graph, and decides which processes should be involved in each step of progressive retry [20]. The communication graph is reset each time a new recovery line is formed.

The following program shows how an application uses the `libft` functions to implement the progressive retry technique.

```
#include <ft.h>
...
main(){
     ...
     setlogfile("examp.log");
     initialization();
     if ((i=in_recovery())>0) {
          recover(INFILE);
          if (i==2)  /* if in step 2 or step 4 */
               ftreorder("examp.log");
     }
     for(;;){
          if (!in_recovery()){
               inso=accept(...);
               ...
          }
          /* receive a message from a client */
          length=ftread(inso,buffer,MAXBUF);
          process_data(newbuf);
          outso=connect_other_process();
```

```
                /* send a message to a server */
                ftwrite(outso,newbuf,MAXBUF,refid);
                ....
            }
    }
```

When the program is in a normal state, it calls `ftread()` to receive and log a message. The message is then processed by calling `process_data()`. The result is stored in `newbuf` and is sent to another process by `ftwrite()`. Function `setlogfile("examp.log")` declares that the file name for the sender log is `examp.log.send` and the file name for the receiver log is `examp.log.recv`.

If the program fails (hangs or crashes) in `process_data()` and is restarted by `watchd`, the return value of function `in_recovery()` becomes 1. In this case, the program restores a checkpointed state by calling the function `recover()`. Then, it calls `ftread()` to read messages from the receiver log file, instead of the regular socket channel. After a logged message is processed, the program calls `ftwrite()` to compare the newly generated message to the corresponding one in the sender log file. If they are identical (deterministic replay), no message is sent. When all of the messages in the receiver log file are replayed successfully, function `in_recovery` returns 0 and the program resumes its normal operation. Otherwise, Step-2 retry is initiated and the program is restarted again. In step 2, the return value of `in_recovery()` becomes 2, and the program reorders the receiver log file by calling function `reorder()` before reexecuting the log. If the program still fails to return to its normal operation, it will then invoke Step-3 retry by also involving some of its senders in the recovery process.

Since checkpointing, messages logging and rollback recovery are provided by `libft` and `watchd`, the actual number of lines of code for implementing progressive retry in an application program is very small. For example, the nDFS file system described in Section 4 consists of more than 60,000 lines of C code; only 10 lines are added and 15 lines are modified to incorporate progressive retry. The performance overhead in this case is measured to be approximately 10%.

6 Extensions and Limitations

In Step-2 retry, we considered all of the messages processed since the most recent checkpoint as possible candidates for reordering. This would in general result in rollback propagation. Another possibility is to allow the reordering of only those messages processed after the latest message was sent [37]. Retry based on message reordering can then be performed locally without affecting any other process. Similarly, in Step-5 retry, we chose the latest lazy-coordinated consistent set of checkpoints for rollback retry; if the penalties associated with revoking outputs (released to outside the system) are large, large-scope rollback by discarding only those logical checkpoints beyond the latest output may be desirable [25].

Although message replaying and message reordering are effective mechanisms for bypassing certain boundary conditions and exceptions, we have also observed failures that cannot be recovered by these two mechanisms. For example, global overload conditions are likely to persist in spite of the retries; exception management messages will again invoke the same faulty exception handler during the retries. Existing solutions include temporarily suspending incoming service requests, deactivating the defective software components, and replacing the faulty software with an earlier stable version. Since these procedures often involve manual operations, they have become the primary sources of extensive system outages and hence the major challenges to achieving high availability.

7 Concluding Remarks

We have described a 5-step progressive retry technique using message logging as well as check-pointing to limit the scope of rollback and thereby provide a means for achieving localized and fast recovery. The technique is designed for continuously-running software systems which can absorb a certain degree of performance overhead and significantly benefit from reduced service unavailability. Our approach has employed message replaying for exploiting piecewise deterministic model, message comparison for validating the above model and message reordering for introducing diversity. Lazy checkpoint coordination has been incorporated to provide consistent sets of checkpoints for the final-step large-scope retry, and can be tuned to accommodate the maximum error latency for each application. Progressive retry has been implemented as part of a Software Fault Tolerance Platform developed at AT&T Bell Laboratories to provide automatic, economical, effective and efficient software failure recovery. Future research includes combining different approaches to software fault tolerance and integrating progressive retry with transaction recovery [38, 39].

Acknowledgement

The authors wish to express their sincere thanks to Andy Lowry (IBM) and Pi-Yu Chung (AT&T) for useful discussions, and to Chia-Mei Chen for her contribution to the implementation.

References

[1] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. San Mateo, CA: Morgan Kaufmann Publishers, 1993.

[2] J. Gray, "A census of tandem system availability between 1985 and 1990," *IEEE Trans. Reliab.*, Vol. 39, No. 4, pp. 409–418, Oct. 1990.

[3] M. Sullivan and R. Chillarege, "Software defects and their impact on system availability - A study of field failures in operating systems," in *Proc. IEEE Fault-Tolerant Computing Symp.*, pp. 2–9, 1991.

[4] D. Jewett, "Integrity S2: A fault-tolerant UNIX platform," in *Proc. IEEE Fault-Tolerant Computing Symp.*, pp. 512–519, 1991.

[5] J. Gray and D. P. Siewiorek, "High-availability computer systems," *IEEE Comput. Mag.*, pp. 39–48, Sept. 1991.

[6] J. Gray, "Dependable systems." *Keynote Speech, 11th Symp. on Reliable Distr. Syst.*, Oct. 1992.

[7] F. Cristian, "Exception handling and software fault tolerance," *IEEE Trans. Comput.*, Vol. C-31, No. 6, pp. 531–540, June 1982.

[8] P. E. Ammann and J. C. Knight, "Data diversity: An approach to software fault-tolerance," in *Proc. IEEE Fault-Tolerant Computing Symp.*, pp. 122–126, 1987.

[9] P. E. Ammann and J. C. Knight, "Data diversity: An approach to software fault-tolerance," *IEEE Trans. Comput.*, Vol. 37, No. 4, pp. 418–425, Apr. 1988.

[10] A. Avizienis, "The N-version approach to fault-tolerant software," *IEEE Trans. Software Eng.*, Vol. SE-11, No. 12, pp. 1491–1501, Dec. 1985.

[11] B. Randell, "System structure for software fault tolerance," *IEEE Trans. Software Eng.*, Vol. SE-1, No. 2, pp. 220–232, June 1975.

[12] E. Adams, "Optimizing preventive service of software products," *IBM J. R&D*, No. 1, pp. 2–14, Jan. 1984.

[13] I. Lee and R. K. Iyer, "Faults, symptoms, and software fault tolerance in the tandem guardian90 operating system," in *Proc. IEEE Fault-Tolerant Computing Symp.*, 1993.

[14] J. F. Bartlett, "A NonStop Kernel," in *Proc. 8th ACM Symp. on Operating Systems Principles*, pp. 22–29, 1981.

[15] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle, "Fault tolerance under UNIX," *ACM Trans. Comput. Syst.*, Vol. 7, No. 1, pp. 1–24, Feb. 1989.

[16] Y. Huang and C. Kintala, "Software implemented fault tolerance: Technologies and experience," in *Proc. IEEE Fault-Tolerant Computing Symp.*, pp. 2–9, June 1993.

[17] B. Bhargava and S. R. Lian, "Independent checkpointing and concurrent rollback for recovery - An optimistic approach," in *Proc. IEEE Symp. Reliable Distributed Syst.*, pp. 3–12, 1988.

[18] Y. M. Wang and W. K. Fuchs, "Optimistic message logging for independent checkpointing in message-passing systems," in *Proc. IEEE Symp. Reliable Distributed Syst.*, pp. 147–154, Oct. 1992.

[19] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. Comput. Syst.*, Vol. 3, No. 1, pp. 63–75, Feb. 1985.

[20] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Trans. Software Eng.*, Vol. SE-13, No. 1, pp. 23–31, Jan. 1987.

[21] K. Li, J. F. Naughton, and J. S. Plank, "Checkpointing multicomputer applications," in *Proc. IEEE Symp. Reliable Distributed Syst.*, pp. 2–11, 1991.

[22] Y. M. Wang and W. K. Fuchs, "Lazy checkpoint coordination for bounding rollback propagation," in *Proc. IEEE Symp. Reliable Distributed Syst.*, pp. 78–85, Oct. 1993.

[23] A. Borg, J. Baumbach, and S. Glazer, "A message system supporting fault-tolerance," in *Proc. 9th ACM Symp. on Operating Systems Principles*, pp. 90–99, 1983.

[24] M. L. Powell and D. L. Presotto, "Publishing: A reliable broadcast communication mechanism," in *Proc. 9th ACM Symp. Oper. Syst. Principles*, pp. 100–109, 1983.

[25] R. E. Strom and S. Yemini, "Optimistic recovery in distributed systems," *ACM Trans. Comput. Syst.*, Vol. 3, No. 3, pp. 204–226, Aug. 1985.

[26] D. B. Johnson and W. Zwaenepoel, "Recovery in distributed systems using optimistic message logging and checkpointing," *J. Algorithms*, Vol. 11, pp. 462–491, 1990.

[27] A. P. Sistla and J. L. Welch, "Efficient distributed recovery using message logging," in *Proc. 8th ACM Symposium on Principles of Distributed Computing*, pp. 223–238, 1989.

[28] T. T.-Y. Juang and S. Venkatesan, "Crash recovery with little overhead," in *Proc. IEEE Int. Conf. Distributed Comput. Syst.*, pp. 454–461, 1991.

[29] E. N. Elnozahy and W. Zwaenepoel, "Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit," *IEEE Trans. Comput.*, Vol. 41, No. 5, pp. 526–531, May 1992.

[30] R. E. Strom, D. F. Bacon, and S. A. Yemini, "Volatile logging in n-fault-tolerant distributed systems," in *Proc. IEEE Fault-Tolerant Computing Symp.*, pp. 44–49, 1988.

[31] Y. M. Wang, A. Lowry, and W. K. Fuchs, "Consistent global checkpoints based on direct dependency tracking." Research Report RC 18465, IBM T.J. Watson Research Center, Yorktown Heights, New York, Oct. 1992.

[32] K. H. Kim, J. H. You, and A. Abouelnaga, "A scheme for coordinated execution of independently designed recoverable distributed processes," in *Proc. IEEE Fault-Tolerant Computing Symp.*, pp. 130–135, 1986.

[33] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel, "The performance of consistent checkpointing," in *Proc. IEEE Symp. Reliable Distributed Syst.*, pp. 39–47, Oct. 1992.

[34] D. Briatico, A. Ciuffoletti, and L. Simoncini, "A distributed domino-effect free recovery algorithm," in *Proc. IEEE 4th Symp. on Reliability in Distributed Software and Database Systems*, pp. 207–215, 1984.

[35] M. Baker and M. Sullivan, "The recovery box: Using fast recovery to provide high availability in the UNIX environment," in *Proc. Summer '92 USENIX*, pp. 31–43, June 1992.

[36] D. Korn, Y. Huang, G. Fowler, and H. Rao, "A user-level replicated file system," in *Proc. Summer '93 USENIX*, pp. 279–290, June 1993.

[37] P. Jalote, "Fault tolerant processes," *Distributed Computing*, Vol. 3, pp. 187–195, 1989.

[38] S. K. Shrivastava, L. V. Mancini, and B. Randell, "The duality of fault-tolerant system structures," *Software - Practice and Experience*, Vol. 23, No. 7, pp. 773–798, July 1993.

[39] L. Strigini and F. D. Giandomenico, "Flexible schemes for application-level fault tolerance," in *Proc. IEEE Fault-Tolerant Computing Symp.*, pp. 86–95, 1991.